R tutorial

MATH 4263/5373: Applied Numerical Analysis

1/31/2022

Coding

In either (base) R or Rstudio, you can save a 'dot R' file containing just plain text code and comments. It will be best to type into a script and execute the code rather than to type commands into the console (similar to a calculator). Working from a script helps you preserve a record of your work. When beginning programming, you should try to do a few steps by hand, in a script to try to recognize the structure of the calculation and build intuition for the process. Then, combined with the pseudo-code and any other materials, try to streamline your work into a functioning program.

The above approach works in (base) R or Rstudio. If you are feeling fancy, you can use the RMarkdown syntax to build a report with embedded code. To begin you would click File > New File > R Markdown, motivation for this approach is described below.

RMarkdown

You can use a script and console to run and save calculations (use it like a calculator, but with better record keeping), or ultimately to write programs and script your function calls to generate output. Ultimately Markdown languages are a powerful communcation and recordkeeping tool - this document was written in RMarkdown, which allows for a combination of computation and formatted typesetting with a hybrid LaTeX language. The downside to this is the simultaneous use and debugging of two languages.

Assignment

You are encouraged to type in commands from the pdf file below (do not copy and paste). You should probably start with a simple plain text script and not an R markdown file.

Calculator functions

We can use R for basic calculator functionality.

```
2+2
```

```
## [1] 4
```

```
sin(2)
```

```
## [1] 0.9092974
```

```
log(10)
```

```
## [1] 2.302585
```

```
exp(1)
```

```
## [1] 2.718282
```

```
5%%2 ## what might this operator '%%' do?
```

```
## [1] 1
```

```
6%%2
```

```
## [1] 0
```

Generating data

We can make vectors with the `c()` command and assign them to a variable with the assignment operator `<-`. Later we will use more advanced commands to read comma-separate files or spreadsheet output.

```
w <- 1:10
z <- seq(0, 10, length=11)
(z <- seq(0, 10, by=0.5)) ## what are the major differences between these lines?
```

```
##  [1]  0.0  0.5  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0
## [16]  7.5  8.0  8.5  9.0  9.5 10.0
```

```
length(z) ## how big?
```

```
## [1] 21
```

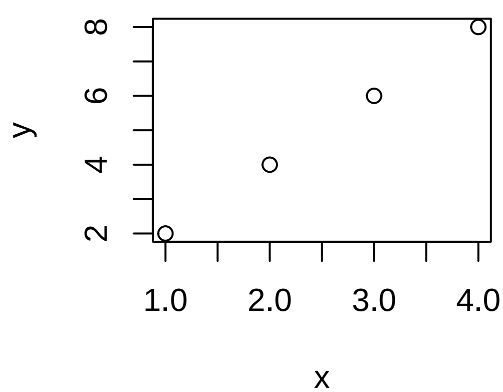
```
rev(z) ## reverse the order of elements
```

```
##  [1] 10.0  9.5  9.0  8.5  8.0  7.5  7.0  6.5  6.0  5.5  5.0  4.5  4.0  3.5  3.0
## [16]  2.5  2.0  1.5  1.0  0.5  0.0
```

```
c(w, z) ## combine objects
```

```
##  [1]  1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0  0.0  0.5  1.0  1.5  2.0
## [16]  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5
## [31] 10.0
```

```
x <- c(1, 2, 3, 4)
y <- 2*x
plot(x,y)
```



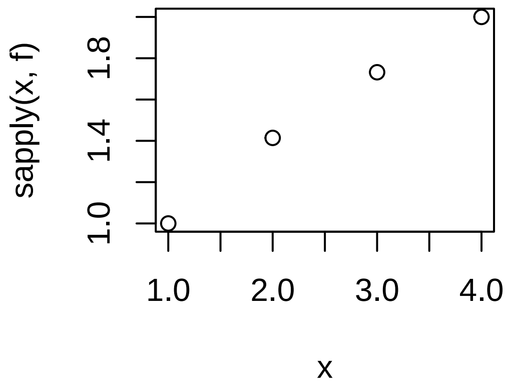
Defining and using functions

We can use a `function(...)` to define mathematical functions or programs. For mathematical functions there are a variety of tools for evaluation.

```
f <- function(x) sqrt(x)
sapply(x, f)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000
```

```
plot(x, sapply(x, f))
```



Basic programming

We can use for loops for automation.

```
for(i in 1:5){
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

We can use for conditionals for control.

```
for(i in 1:5){
  if(i%%2 == 1){ ## note the == for equality testing
    print(i^2)
  }else{
    print(i^3)
  }
}
```

```
## [1] 1
## [1] 8
## [1] 9
## [1] 64
## [1] 25
```

Putting it all together

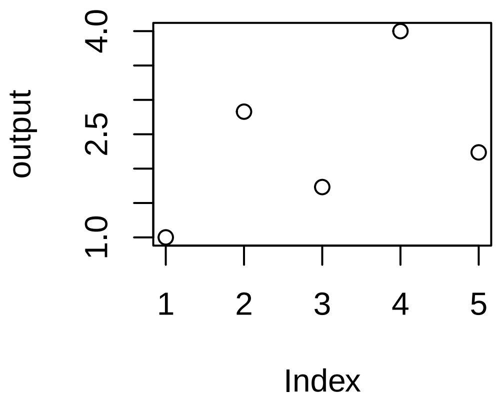
Suppose we wanted to run the following small program and store the results for later use.

```
output <- NULL
for(i in 1:5){
  if(i%%2==1){
       output <- c(output, f(i)) ## odd i
  }else{
       output <- c(output, 2*f(i)) ## even i
  }
```

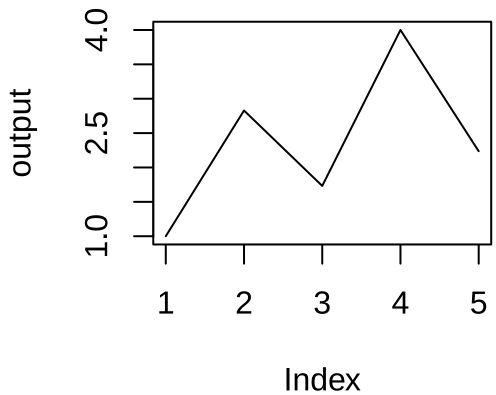
```
}
output
```

```
## [1] 1.000000 2.828427 1.732051 4.000000 2.236068
```

```
plot(output)
```




```
plot(output, type='l')
```



Putting it all together (and more)

Suppose we wanted to run the following small program and store the results for later use and do some work with the output.

```
prog <- function(N){ ## function 'prog' has argument N
output <- NULL       ## initialize storage
for(i in 1:N){
  if(i%%2==1){       ## sample logic
       output <- c(output, f(i))  ## sample storage
  }else{
       output <- c(output, 2*f(i))
  }
}
return(output)       ## return result
}

out <- prog(100)     ## execute program, store result
head(out)
```

```
## [1] 1.000000 2.828427 1.732051 4.000000 2.236068 4.898979
```

```
tail(out)
```

```
## [1]  9.746794 19.595918  9.848858 19.798990  9.949874 20.000000
```

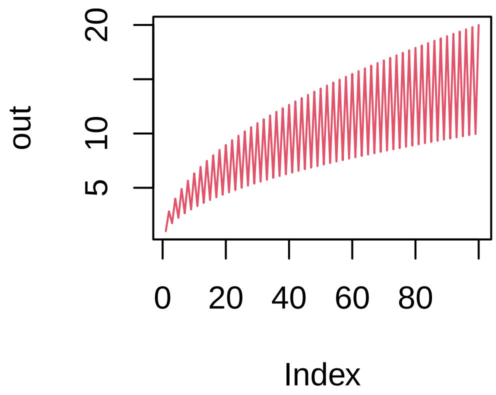
```
min(out)
```

```
## [1] 1
```

```
max(out)
```

```
## [1] 20
```

```
plot(out, type='l', col=2)
```

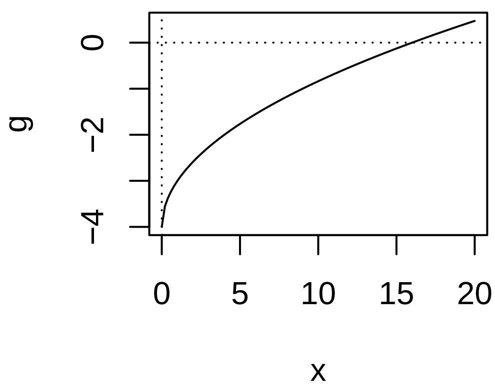


Built-in functions

R has a variety of built-in commands for our current and future needs. We want to build these capabilities ourselves, but it is good to know about what is available.

```
g <- function(x) f(x) - 4

plot(g, xlim=c(0, 20))
abline(h=0, lty=3)
abline(v=0, lty=3)
```




```
uniroot(g, lower=0, upper = 20) ## based on bisection
```

```
## $root
## [1] 16
##
## $f.root
## [1] 0
```

```
##
## $iter
## [1] 4
##
## $init.it
## [1] NA
##
## $estim.prec
## [1] 7.055728
```

```
root <- uniroot(g, lower=0, upper = 20)$root
?uniroot
```

Challenges

Arithmetic

Experiment with the commands for manipulating numerical values. Exploring the help menu might show you related commands.

```
pi
```

```
## [1] 3.141593
```

```
ceiling(pi)
```

```
## [1] 4
```

```
floor(pi)
```

```
## [1] 3
```

```
trunc(pi)
```

```
## [1] 3
```

```
round(pi, 5)
```

```
## [1] 3.14159
```

```
signif(pi, 3)
```

```
## [1] 3.14
```

```
signif(pi - 3, 3)
```

```
## [1] 0.142
```

How do these differ if we use $-\pi$ rather than π?

Rootfinding

Try a few steps of the bisection method by hand. Define a and b and your function f. Start by storing approximations manually, $p0, p1, \ldots$, but consider how you might streamline your scratchwork by using a loop. After that you might wrap a function on the outside that accepts parameters, or you might practice using print statements to return information to the screen. This might be a good place to review the script posted to D2L.

```
for(i in 1:5){
  #print(c(i,i^2)) ## unformatted
```

```r
  print(paste("For i=",i,", the value i^2=", i^2, ".", sep=''))
}
```

```
## [1] "For i=1, the value i^2=1."
## [1] "For i=2, the value i^2=4."
## [1] "For i=3, the value i^2=9."
## [1] "For i=4, the value i^2=16."
## [1] "For i=5, the value i^2=25."
```

```r
for(i in 1:5){
  #print(c(i, i^2)) ## unformatted
  cat("For i=",i,", the value i^2=", i^2, ".\n",sep='')
}
```

```
## For i=1, the value i^2=1.
## For i=2, the value i^2=4.
## For i=3, the value i^2=9.
## For i=4, the value i^2=16.
## For i=5, the value i^2=25.
```